COMP 520 - Compilers

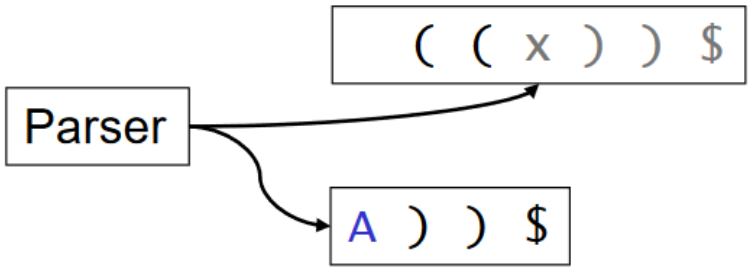
Lecture 04 – Lexers, Grammar Properties, Transformations

Is miniJava LL(1)?

- Why or why not?
- What are some languages that are LL(1)?

Previous Lecture

- Leftmost derivation
- Top-down parsing with a PDA
- Recursive descent parsing



CFG G_0

S ::= A $

A ::= (A)

A ::= x

```
parseS() {
    parseA();
    accept( '$' );
}

parseA() {
    if ( currChar == '(' ) {
        accept( '(' );
        parseA();
        accept( ')' );
    }
    else
        accept( 'x' );
}
```

Previous Lecture (2)

- LL(1) condition
  - Parser can always predict using the next **(1)** input token
  - Reading **L**eft to Right
  - Using **L**eftmost derviation

Final Exam Date Posted

- On the syllabus
- Thursday May 9th, 2024

Lexical Analysis Theory

Scanners (Lexers)

- Purpose: extract Tokens from a character stream
- Recall: a Token is a terminal symbol in the parser grammar

- Examples: A number, identifier, operator, or keyword

Scanning Approach

Similar to parsing with some key exceptions:

- The scanner grammar is simple.
- Terminals are individual characters
- Non-terminals are the terminals (Tokens) of the Parser

Scanning Approach

Similar to parsing with some key exceptions:

- The scanner grammar is simple.
- Terminals are individual characters
- Non-terminals are the terminals (Tokens) of the Parser

- The rule for each non-terminal is a regular expression
  - Thus no need for recursion in scanner grammar

Whitespace in the Scanner

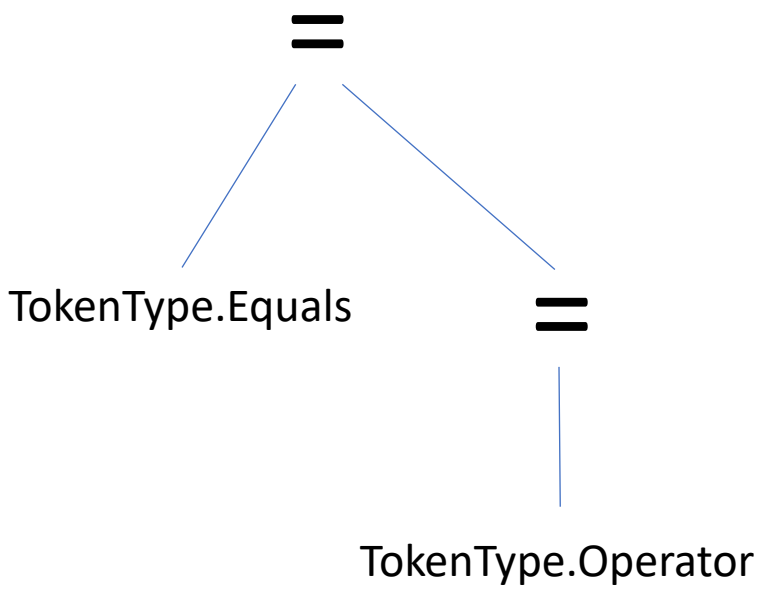
- Consider the two character == operator
- Should each = be a Token and then the Parser check for "==" by looking for and accepting two = Tokens?

Handling multi-character Tokens

- Note, '=' is not an operator, but instead used only for assignment as a standalone statement or local variable initialization (not as a part of an expression) in miniJava
- Thus as an example,
  - '=' is a TokenType.Equals while '==' is TokenType.Operator
- So the scanner has a choice to make:

Handling multi-character Tokens

- So the scanner has a choice to make:



End of Tokens

- Lastly, the Scanner must have an elegant way of conveying the source file has no more Tokens to produce.

- Can be done by returning a null Token, or a TokenType.EOT, or however you want to code the `scan` method.

Scanner Grammar

- By looking at leaf nodes in the Grammar graph, we can find our Tokens.

- Token ::= Operator | Number | Identifier | Keyword

- Number ::= Digit | Number Digit
- Identifier ::= Alpha | Identifier AlphaNumUnderscore
- Keyword ::= `while` | `class` | `public` | …
- Note: there is a problem!

Hold on!

- Those grammar rules do not look like regular expressions:
  - Number ::= Digit | Number Digit
  - Identifier ::= Alpha | Identifier AlphaNumUnderscore

- They use recursion!
- Can we translate this grammar?

EBNF Grammars

Extended Backus-Naur Form

Backus-Naur Form

CFG has a rule in the form $A ::= \alpha$ where $A \in N$

Sequence α can contain:

- Sequences of terminals and non-terminals ($T \cup N$)
  - IfStmt ::= `if` Exp `then` Stmt ElsePart
  - SkipStmt ::= `skip`
- Empty Sequence ε

Extended Backus-Naur Form

Sequence α can contain:

- Sequences allowed in BNF
- Choice **|**
  - ElsePart ::= `else` Stmt **|** ε
- Repetition ***
  - BlockStmt ::= { Statement* }
- Grouping **()**
  - ClassDeclaration ::= `class id` { **(**FieldDeclaration|MethodDeclaration**)*** }

More EBNF Rules

- If interested in EBNF grammars:
  - https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

New Sentence Generation Rule

- A sentence w is generated by EBNF Grammar G if:
  - Recall: $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$, where $\alpha_n = \text{w}$ and $S = \alpha_1$
  - S is the start symbol
  - w consists exclusively of terminals
  - $\alpha_i \Rightarrow \alpha_{i+1}$ if
    - $\alpha_i = \beta \mathbf{W} \gamma$ and $\alpha_{i+1} = \beta \boldsymbol{\mu} \gamma$
    - **W** ::= ω is a rule in G
    - **NEW RULE: ???**
- EBNF grammars G generate L(G), and **L(G) is a CFG**

New Sentence Generation Rule

- A sentence w is generated by EBNF Grammar G if:
  - Recall: $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$, where $\alpha_n = \text{w}$ and $S = \alpha_1$
  - S is the start symbol
  - w consists exclusively of terminals
  - $\alpha_i \Rightarrow \alpha_{i+1}$ if
    - $\alpha_i = \beta \mathbf{W} \gamma$ and $\alpha_{i+1} = \beta \boldsymbol{\mu} \gamma$
    - **W** ::= ω is a rule in G
    - **Regular expression $\boldsymbol{\omega}$ can generate $\boldsymbol{\mu}$**
- EBNF grammars G generate L(G), and **L(G) is a CFG**

EBNF

- EBNF is a convenience
  - Any EBNF can be rewritten as a CFG

- Example: Eliminate EBNF extensions in this rule:

ArgumentList ::= Expression (,Expression)*

EBNF

- EBNF is a convenience
  - Any EBNF can be rewritten as a CFG

- Example: Eliminate EBNF extensions in this rule:

ArgumentList ::= Expression (,Expression)*

-=-=-=-=-=-=-=-=-=-=-=-=-

ArgumentList ::= Expression

ArgumentList ::= Expression, ArgumentList

EBNF Benefits

Why?

- Simpler expression of grammars
- Better target for grammar transformations

- Most importantly: Much easier to write recursive descent parsers once you have transformed a grammar into EBNF

Grammar Transformations

Substitution method

CFG

C ::= A b D

A ::= c | d

EBNF

?

Substitution method

CFG

C ::= A b D

A ::= c | d

EBNF

C ::= (c | d) b D

Left-Factorization

Original

IfStmt ::= if Exp then Stmt
| if Exp then Stmt else Stmt

Easier to parse and convenient

IfStmt ::= if Exp then Stmt
(ε | else Stmt)

Remove Left Recursion

Original

N ::= X | NY

Easier to parse and convenient

N ::= X (Y)*

Question: Can you do the same below? If so, how?
Reference ::= id | this | Reference . id

Simplification Example

- This grammar is not very easy to read:
  - E ::= T | E Op T
  - T ::= (E) | num
  - Op ::= + | ×

Simplification Example

- This grammar is not very easy to read:
  - E ::= T | E Op T
  - T ::= (E) | num
  - Op ::= + | ×
- Remove left recursion
  - **E ::= T (Op T)***
  - T ::= (E) | num
  - Op ::= + | ×

Simplification Example

- This grammar is not very easy to read:
  - E ::= T | E Op T
  - T ::= (E) | num
  - Op ::= + | ×
- Remove left recursion
  - E ::= T (**Op** T)*
  - T ::= (E) | num
  - **Op ::= + | ×**
- Substitute Op for E
  - E ::= T ((+|×) T)*

Simplification Example

- This grammar is not very easy to read:
  - E ::= T | E Op T
  - T ::= (E) | num
  - Op ::= + | ×
- Remove left recursion
  - E ::= T (Op T)*
  - T ::= (E) | num
  - Op ::= + | ×
- Substitute Op for E
  - E ::= T ((+|×) T)*

Much Easier Result:

E ::= T ((+|×) T)*

T ::= (E) | num

Exercise- Simplify these, and are they LL(1)?

Right recursion

E ::= T | T Op E

T ::= (E) | num

Op ::= + | ×

Left and Right recursion

E ::= T | E Op E

T ::= (E) | num

Op ::= + | ×

EBNF and Recursive Descent

How exactly is EBNF helpful for RD? Let's explore!

The Basics for EBNF and RD

Choice

- Conditional: $\alpha \mid \beta$
- Implemented with an if statement
- If LL(1), only need to check current Token

Repetition

- Repetition: α^*
- Implemented with a while statement
- Condition depends on the starter set for α

The Basics for EBNF and RD

Choice

- Conditional: $\alpha \mid \beta$
- Implemented with an if statement
- If LL(1), only need to check current Token

Repetition

- Repetition: α^*
- Implemented with a while statement
- Condition depends on the starter set for α

Consider the Example from Earlier

S ::= E $

E ::= T ((+ | ×) T)*

T ::= (E) | num

num = {0, 1, …, 9}

Consider the Example from Earlier: Solution

S ::= E $

E ::= T ((+ | ×) T)*

T ::= (E) | num

num = {0, 1, …, 9}

```
void parseS() {
   parseE();
   accept('$');
}
void parseE() {
   parseT();
   while (currChar == '+'
             || currChar == '×') {
         acceptIt();
         parseT();
   }
}
void parseT() {
   switch (currChar) {
      case '0': … case '9':
            acceptIt();
            return;
      case '(':
            acceptIt();
            parseE();
            accept( ')' );
            return;
      default:
             error();
   }
}
```

Consider the Example from Earlier: Solution

S ::= E $

E ::= T ((+ | ×) T)*

T ::= (E) | num

num = {0, 1, …, 9}

It's like it writes itself!

```
void parseS() {
   parseE();
   accept('$');
}
void parseE() {
   parseT();
   while (currChar == '+'
             || currChar == '×') {
         acceptIt();
         parseT();
   }
}
void parseT() {
   switch (currChar) {
      case '0': … case '9':
            acceptIt();
            return;
      case '(':
            acceptIt();
            parseE();
            accept( ')' );
            return;
      default:
             error();
   }
}
```

Helpful Definitions and Properties for EBNF Grammars

One Rule to Ring them all Together

- Given an EBNF grammar..
- Multiple rules with the same non-terminal can be combined

E.g.

- A ::= α_1
- A ::= α_2
- A ::= α_3

- A ::= $\alpha_1 | \alpha_2 | \alpha_3$

Nullable

- Nullable[α] $\equiv$ Can α derive the empty string?

- Examples:
  - Visibility ::= (`public`|`private`)?
  - BlockStmt ::= `{` Statement* `}`

Starters

- Starters[α] is a set of terminals that indicate the current sequence could be α
- Includes ε if Nullable[α]

- What is Starters[Reference]?
  - Reference ::= `id` | `this` | Reference `.` `id`

Followers

- Followers(A) where $A \in N$
- .. is a set of terminals that may follow A

- That means after parsing A, what terminals occur afterwards?

- Note, there exists a form of augmented grammars where ONLY Followers(S) can include ε

More EBNF Thursday, Back to Lexical Analysis

Remove Recursion

- These was previously an issue:
  - Number ::= Digit | Number Digit
  - Identifier ::= Alpha | Identifier AlphaNumUnderscore

- With our grammar techniques, can we remove recursion?

Remove Recursion

- These was previously an issue
  - Number ::= Digit | Number Digit
  - Identifier ::= Alpha | Identifier AlphaNumUnderscore

- With our grammar techniques, can we remove recursion?

YES!

- Number ::= Digit (Digit)*
- Identifier ::= Alpha (AlphaNumUnderscore)*

Lexical Parsing – Longest Match

- Generally speaking, choosing the longest Token often works best
  - Not always true, there are some crazy examples
- E.g., pick >= over >
- So even if your language is LL(1), your Scanner might be hiding some lexical parsing complexities!

Lexical Parsing – LL(1) does not apply

- The Scanner may be hiding some "**look more than 1 character ahead**," but that appears unavoidable for most languages.
- Consider **IntLiteral / FloatLiteral** (note **FloatLiteral** is not in miniJava)

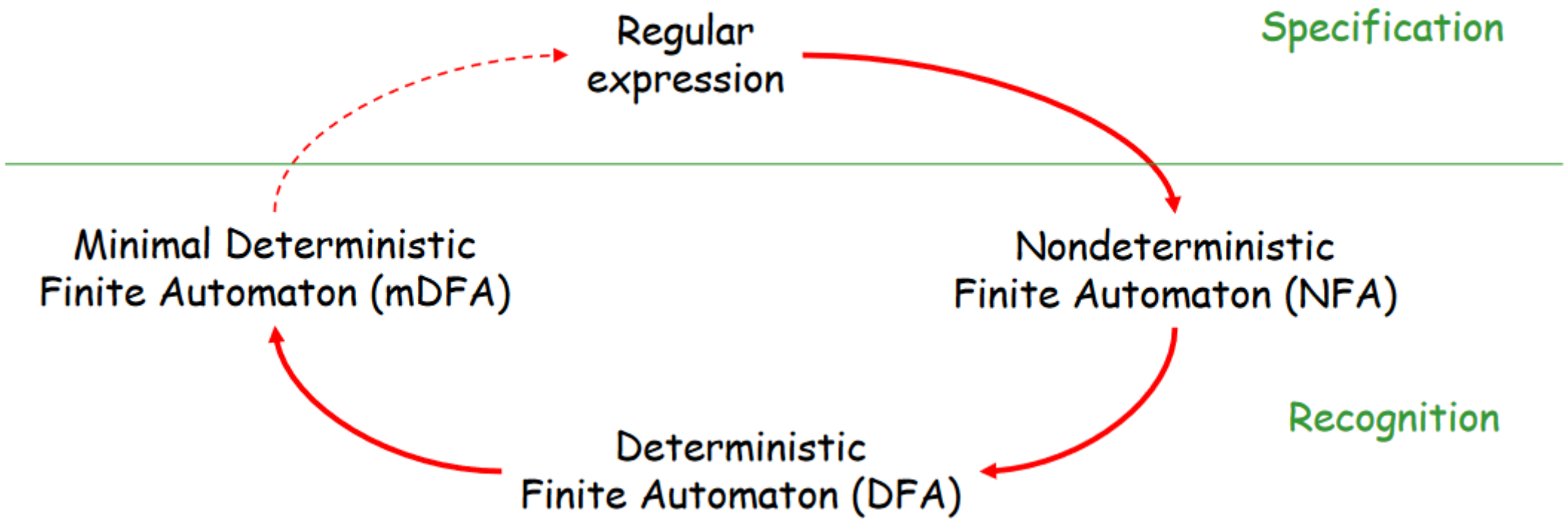
```
if( currentChar in [ '0' … '9' ] ) {
    Accumulate letters until non-digit
    if( currentChar !=  '.'  ) return makeToken(IntLiteral);
    Accumulate letters until non-digit
    return makeToken(FloatLiteral);
}
```

How do we construct a Scanner?

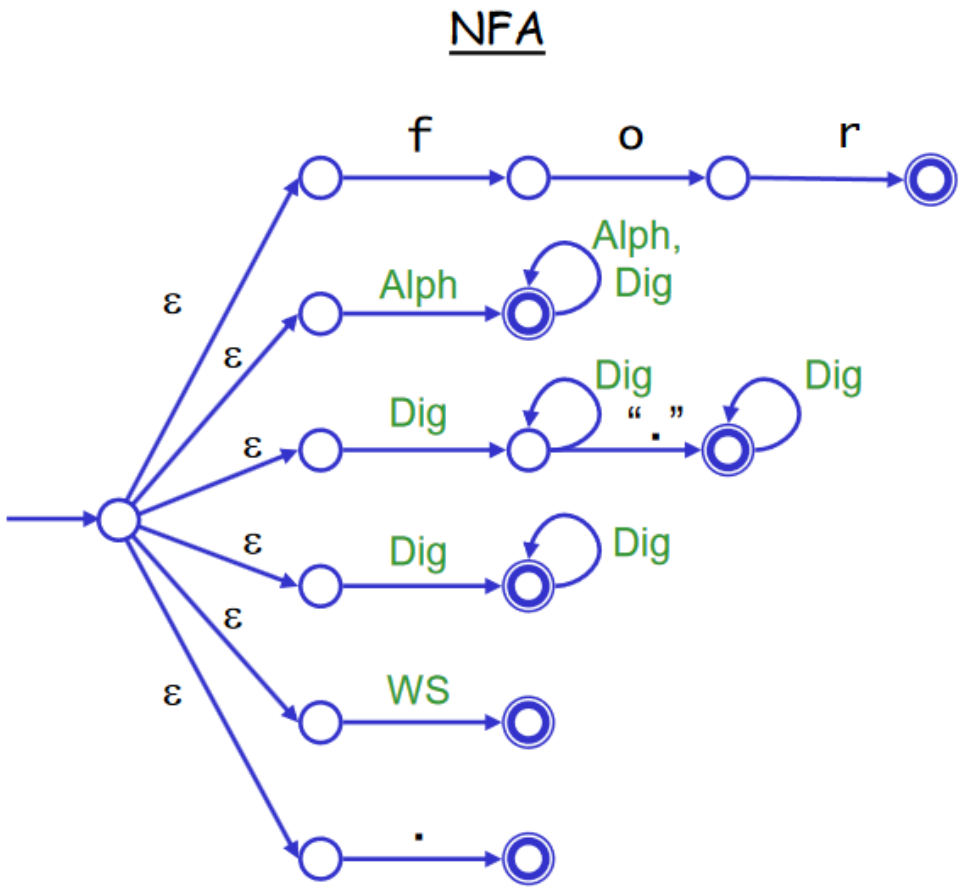
- Can actually be systematically generated
- Construct a finite automaton from the specifications
- The scanner will become the implementation of the finite automaton
- Correctness is guaranteed!

- Note: use of a scanner/lexer generator not allowed for PA1, unless you construct it yourself

Systematic Scanner Generator – Jan Prins



Non-deterministic Generation - Jan Prins



Token	Regular Expr
FOR	“for”
ID	Alpha AlphaNum*
REAL	Int ”.” Int?
INT	Int
(no token)	WS
ERR	. (meaning any char)

How do we implement an NFA?

- Side question: If we want to implement something non-deterministic, what kind of computational complexity are we looking at?

How do we implement an NFA?

- Side question: If we want to implement something non-deterministic, what kind of computational complexity are we looking at?

- Exponential, so not very efficient!

NFA General Idea

- S= a set of possible states where Q= total number of states
- q= arbitrary state, $\delta(q, a)$= transition function from q given input a

- Define:
  - ε-closure(S) = $\bigcup_{q \in S}$(states reachable from q via ε edges)
  - $\hat{\delta}(S, a) = \bigcup_{q \in S} \delta(q, a)$

- Note: ε-closure can be as large as $2^Q \rightarrow 2^Q$

Initialize

- Current state= S
- Initialize with initial state q_0

- S = ε-closure(q_0)

NFA Loop

- Where a = current char, F= final states

while(S $\neq \emptyset$)

- Update S := ε-closure($\hat{\delta}(S, a)$)
  - For all possible states we are in, what states can we reach with a?
- Update a := next_char()
- If $S \cap F \neq \emptyset$, then some final state is reachable, and our Token could be min($S \cap F$)

When $S = \emptyset$ (cannot reach states), restore to last valid state S and return a Token.

Generate DFA from Rules

- To generate a DFA, locate and "organize" rules.
- Order rules $A_1,\ldots,A_n$ where if rule A_i applies, then A_{i+1} cannot.

- If rules contain the same Starters, then combine those rules and construct your DFA s.t. the differentiation can be specified later as a branch.
  - (See example of IntLiteral / FloatLiteral)

Thursday 1/25

- More on Nullable, Starter, and Followers
- Derivation of prediction sets
- Finish Scanner/Parser correctness

End